PROFESSOR: So today is our third and probably final lecture on approximation algorithms. We're going to take a different approach to proving inapproximability to optimization problems called gap problems. And we'll think about gap-preserving reductions. We will prove along the way optimal lower bound on MAX-3SAT and other fun things.

So let's start with what a gap problem is. I haven't actually seen a generic definition of this, so this is new terminology, but I think helpful. A gap problem is a way of converting an optimization problem into a decision problem. Now, we already know one way to do that. If you have an NPO optimization problem, you convert it into the obvious decision problem, which is OPT at most k. For the minimization problem, that is NP-complete.

But we want to get something useful from an approximability standpoint. And so the idea is, here's a different problem. Instead of just deciding whether OPT is at most k, we want to distinguish between OPT being at most k versus OPT being at least k over c for some value c. And the analogy here is with c approximation algorithm, c does not have to be constant, despite the name. Could be a function of n. Maybe it's log n. Maybe it's n to the epsilon, whatever.

So for a minimization problem, normally-- did I get this the right way? Sorry, that should be c times k. For minimization and for maximization, it's going to be the reverse. And I think I'm going to use strict inequality here also.

So a minimization. There's a gap here between k and c times k. We're imagining here c is bigger than 1. So distinguishing between being less than k and being at least c times k leaves a hole. And the point is you are promised that your input-- you have a question?

AUDIENCE: The second one, should it really be c over k?

PROFESSOR: Sorry, no. Thank you. C and k sound the same, so it's always easy to mix them up. Cool. So the idea is that you're-- so in both cases, there's a ratio gap here of c. And the idea is that you're promised that your input falls into one of these two categories. What does it mean to distinguish-- I mean, I tell you up front the input either has this property or this property, and I want you to decide which one it is. And we'll call these yes instances and these no instances.

Normally with a decision problem, the no instance is that OPT is just one bigger or one smaller. Now we have a big gap between the no instances and the yes instances. We're told that that's true. This is called promise problem. And effectively what that means is if you're trying to come up with an algorithm to solve this decision problem, you don't care what the algorithm does if OPT happens to fall in between.

The algorithm can do whatever it want. It can output digits of pi in the middle, as long as when OPT is it at most k or greater than or equal to k, it outputs yes. And when OPT is at least a factor of c away from that, it outputs no. So that's an easier problem. And the cool thing is if the c gap version of a problem is NP-hard, then so is c approximating the original problem. So this really is in direct analogy to c approximation.

And so this lets us think about an NP hardness for a decision problem and prove an inapproximability result. This is nice because in the last two lectures, we were having to keep track of a lot more in just defining what inapproximability meant and APX hardness and so on. Here it's kind of back to regular NP hardnes. Now, the techniques are completely different in this world than our older NP hardness proofs. But still, it's kind of comforting to be back in decision land.

Cool. So because of this implication, in previous lectures we were just worried about proving inapproximability. But today we're going to be thinking about proving that gap problems are NP-hard, or some other kind of hardness. This is a stronger type of result. So in general, inapproximability is kind of what you care about from the

algorithmic standpoint, but gap results saying that hey, your problem is hard even if you have this huge gap between the yes instances and the no instances, that's also of independent interest, more about the structure of the problem.

But one implies the other. So this is the stronger type of thing to go for. The practical reason to care about this stuff is that this gap idea lets you get stronger inapproximability results. The factor c you get by thinking about gaps in practice seems to be larger than the gaps you get by L reductions and things.

So let me tell you about one other type of gap problem. This is a standard one, a little bit more precise. Consider MAX-SAT. Pick your favorite version of MAX-SAT, or MAX-CSP was the general form where you could have any type of clause. Instead of just a c gap, we will define slightly more precisely an a, b gap, which is to distinguish between OPT is less than a times the number of clauses, and OPT is at least b times the number of clauses.

So whereas here everything was relative to some input k that you want to decide about, with SAT there's a kind of absolute notion of what you'd like to achieve, which is that you satisfy everything, all clauses are true. So typically we'll think about b being one. And so you're distinguishing between a satisfiable instance where all clauses are satisfied, and something that's very unsatisfiable. There's some kind of usually constant fraction unsatisfiable clauses. We need this level of precision thinking about when you're right up against 100% satisfied versus 1% satisfied or something like that, or 1% satisfiable. Cool.

AUDIENCE: Do you use the same notation for one [INAUDIBLE] or only for-- so the one problem like the maximum number of ones you can get.

PROFESSOR: I haven't seen it, but yeah, that's a good question. Certainly valid to do it for-- we will see it for one other type of problem. For any problem, if you can define some absolute notion of how much you'd like to get, you can always measure relative to that and define this kind of a, b gap problem. Cool.

All right. So how do we get these gaps? There's two, well maybe three ways, I

guess. In general, we're going to use reductions, like always. And you could start from no gap and make a gap, or start from a gap of additive one and turn it into a big multiplicative gap. That will be gap-producing reductions.

You could start with some gap and then make it bigger. That's gap-amplifying reduction. Or you could just start with a gap and try to preserve it. That would be gap-preserving reductions. In general, once you have some gap, you try to keep it or make it bigger to get stronger hardness for your problem.

So the idea with a gap-producing reduction is that you have no assumption about your starting problem. In general, reduction we're going from some problem a to some problem b. And what we would like is that the output instance to problem b, the output of the reduction has OPT equal to k or, for a minimization problem, OPT bigger than c times k.

And for a maximization problem, OPT less than k over c. So that's just saying we have a gap in the output. We assume nothing about the input instance. That would be a gap-producing production. Now we have seen some of these before, or at least mentioned them. One of them, this is from lecture three, I think for Tetris. We proved NP hardness, which was this three partition reduction. And the idea is that if you could satisfy that and open this thing, then you could get a zillion points down here.

In most of the instances down here, we squeeze this down to like an n to the epsilon. That's still hard. And so n to the 1 minus epsilon of the instances down here, and you're given a ton of pieces to fill in the space, get lots of points. If the answer was no here, then you won't get those points. And so OPT is very small, at most, say, n to the epsilon. If you can solve this instance, we have a yes instance in the input, then we get end points. So the gap there is n to the 1 minus epsilon.

So the Tetris reduction, we assume nothing about the three partition instance. It was just yes or no. And we produced an instance that had a gap of n to the 1 minus epsilon. We could set epsilon to any constant we want bigger than zero.

We also mentioned another such reduction. And in general, for a lot of games and puzzles, you can do this. It's sort of on all or nothing deal. And gap-producing reduction is a way to formalize that. Another problem we talked about last class I believe was non-metric TSP. I just give you a complete graph. Every edge has some number on it that's the length of that edge.

You want to find a TSP tour of minimum total length. This is really hard to approximate because depending on your model, you can use let's say edge weights. And to be really annoying would be 0, comma 1. And if I'm given a Hamiltonicity instance, wherever there's an edge, I put a weight of zero. Wherever there's not an edge, I put a weight of one.

And then if the input graph was Hamiltonian, it's a yes instance. Then the output thing will have a tour of length zero. And if the input was not Hamiltonian, then the output will have weight n. Ratio between n and zero is infinity. So this is an infinite gap creation if you allow weights of zero.

If you say zero is cheating, which we did and some papers do, you could instead do one and infinity, where infinity is the largest representable number. So that's going to be something like 2 to the n if you allow usual binary encodings of numbers. If you don't, the PB case, then that would be n to some constant. But you get a big gap in any case. So you get some gap equals huge.

So these are kind of trivial senses of inapproximability, but hey, that's one way to do it. What we're going to talk about today are other known ways to get gap production that are really cool and more broadly useful. This is useful when you have a sort of all or nothing problem. A lot of the time, it's not so clear. There's a constant factor approximation. So some giant gap like this isn't going to be possible, but still gaps are possible.

Now, an important part of the story here is the PCP theorem. So this is not about drugs. This is about another complexity class. And the complexity class is normally written PCP of order log n, comma order one. I'm going to simplify this to just PCP as the class. The other notions make sense here, although the parameters don't

turn out to matter too much. And it's rather lengthy to write that every time. So I'm just going to write PCP.

Let me first tell you what this class is about briefly, and then we'll see why it's directly related to gap problems, hence where a lot of these gap-producing reductions come from. So PCP stands for Probabilistically Checkable Proof. The checkable proof refers to NP. Every yes instance has a checkable proof that the answer is yes.

Probabilistically checkable means you can check it even faster with high probability. So normally to check a proof, we take polynomial time in NP. Here we want to achieve constant time. That's the main idea. That can't be done perfectly, but you can do it correctly with high probability.

So in general, a problem in PCP has certificates of polynomial length, just like NP. And we have an algorithm for checking certificates, which is given certificate, and it's given order log n bits of randomness. That's what this first parameter refers to, is how much randomness the algorithm's given. So we restrict the amount of randomness to a very small amount.

And it should tell you whether the instance is a yes instance or a no instance, in some sense if you're given the right certificate. So in particular, if the instance was a yes instance-- so this is back to decision problems, just like NP. There's no optimization here. But we're going to apply this to gap problems, and that will relate us to optimization.

So let's say there's no error on yes instances, although you could relax that. It won't make a big difference. So if you have a yes instance, and you give the right certificate-- so this is for some certificate-- the algorithm's guaranteed to say yes. So no error there. Where we add some slack is if there's a no instance.

Now normally in NP for a no instance, there is no correct certificate. Now, the algorithm will sometimes say yes even if we give it the wrong certificate. There is no right certificate. But it will say so with some at most constant probability. So let's say

the probability that the algorithm says no is at least some constant, presumably less than one. If it's one, then that's NP.

If it's a half, that would be fine. A tenth, a hundredth, they'll all be the same. Because once you have such an algorithm that achieves some constant probability, you could apply it log 1 over epsilon times. And we reduce the error to epsilon. The probability of error goes to epsilon if we just repeat this log 1 over epsilon times. So in constant time-- it didn't say. The order one here refers to the running time of the algorithm. So this is an order one time algorithm.

So the point is, the algorithm's super fast and still in constant time for constant epsilon. You can get arbitrarily small error probability, say one in 100 or one in a million, and it's still pretty good. And you're checking your proof super, super fast. Question.

AUDIENCE: Why is there a limit on the randomness?

PROFESSOR: This limit on randomness is not strictly necessary. For example, n bits of randomness turned out not to help you. That was proved later. But we're going to use this in a moment. It will help us simulate this algorithm without randomness, basically. Yeah.

AUDIENCE: If the verifier runs in constant time, can it either read or was written?

PROFESSOR: So this is constant time in a model of computation where you can read log n bits in one step. So your word, let's say, is log n bits long. So you have enough time to read the randomness. Obviously you don't have time to read the certificate, because that has polynomial length. But yeah, constant time.

Cool. Other questions? So that is the definition of PCP. Now let me relate it to gap problems. So let's say first claim is that if we look at this gap sap problem, where b equals one and a is some constant, presumably less than one, then-- in fact, that should be less than one. Why did I write strictly less than 1 here? This is a constant less than one.

Then I claim that problem is in PCP, that there is a probabilistically checkable proof for this instance. Namely, it's a satisfying variable assignment. Again, this instance either has the prop-- when in a yes instance all of the entire thing is satisfiable. So just like before, I can have a certificate, just like an NP satisfying assignment to the variables is good. In the no instance, now I know, let's say, at most half of the things are satisfied if this is one half.

And so what is my algorithm going to do? In order to get some at most constant probability of failure, it's going to choose a random clause and check that it was satisfied. Uniform random. So I've got log n bits. Let's say there are n clauses. So I can choose one of them at random by flipping log n coins. And then check so that involves-- this is three SATs. It only involves three variables. I check those three variable value assignments in my certificate by random access into the certificate. In constant time, I determine whether that clause is satisfied. If the clause is satisfied, algorithm returns yes. Otherwise, return no.

Now, if it was a satisfying assignment, the algorithm will always say yes. So that's good. If it was not satisfiable, we know that, let's say at most half of the clauses are satisfiable. Which means in every certificate, the algorithm will say no at least half the time. And half is whatever that constant is. So that means the probability that the algorithm is wrong is less than 1 over the gap, whatever that ratio is. Cool? Yeah.

AUDIENCE: So does this [INAUDIBLE]? So for [INAUDIBLE].

PROFESSOR: Let me tell you, the PCP theorem is that NP equals PCP. This is proved. So all problems are in PCP. But this is some motivation for where this class came from. I'm not going to prove this theorem. The original proof is super long. Since then, there have been relatively short proofs. I think the shortest proof currently is two pages long. Still not going to prove it because it's a bit beside the point to some extent. It does use reductions and gap amplification, but it's technical to prove it, let's say.

But I will give you some more motivation for why it's true. So for example, so here's one claim. If one-- let's change this notation. If less than 1, comma 1, gap 3SAT is

NP-hard, then NP equals PCP. So we know that this is true, but before we know that here-- so we just proved that this thing is NPCP. And if furthermore this problem-- we're going to prove this is NP-hard. That's the motivation.

If you believe that it's NP-hard, then we know all problems in NP can reduce to this thing. And then that thing is NPCP. So that tells us that all problems in NP, you can convert them into less than 1, comma 1 gap 3SAT and then get a PCP algorithm for them. So that would be one way to prove the PCP theorem. In fact, the reverse is also true. And this is sort of more directly useful to us.

If, let's say, 3SAT is NPCP, then the gap version of 3SAT is NP-hard. This is interesting because-- this is true because NP equals PCP, in particular 3SAT is NPCP. And so we're going to be able to conclude, by a very short argument, that the gap version of 3SAT is also NP-hard. And this proves constant factor inapproximability of 3SAT. We will see a tighter constant in a little bit, but this will be our first such bound. And this is a very general kind of algorithm. It's kind of cool.

So PCP is easy for the gap version of 3SAT. But suppose there was a probabilistically checkable proof for just straight up 3SAT when you're not given any gap bound, which is true. It does exist. So we're going to use that algorithm. And we're going to do a gap-preserving reduction.

The PCP algorithm we're given, because we're looking at PCP log n, comma order one, runs in constant time. Constant time algorithm can't do very much. In particular, I can write the algorithm as a constant size formula. It's really a distribution over such formulas defined by the log n and the bits. But let's say it's a random variable where for each possible random choice is a constant size formula that evaluates to true or false, corresponding to whether the algorithm says yes or no. We know we can convert algorithms to formulas if they're a short amount of time.

So we can make that a CNF formula. Why not? 3CNF if we want. My goal is to-- I want to reduce 3SAT to the gap version of 3SAT. Because 3SAT we know is NP-hard. So if I can reduce it to the gap version of 3SAT, I'm happy. Then I know the

gap version of 3SAT is also hard.

So here is my reduction. So I'm given the 3SAT formula, and the algorithm evaluates some formula on it and the certificate. What I'm going to do is try all of the random choices. Because there's only log n bits, there's only polynomially many possible choices for those bits. Order log n so it's n to some constant.

And I want to take this formula, take the conjunction over all of those choices. If the algorithm always says yes, then this formula will be satisfied. So in the yes instance case, I get a satisfiable formula. So yes, complies satisfiable, 100% satisfiable. That corresponds to this number. I want it to be 100% in the yes case.

In the no case, I know that a constant fraction of these random choices give a no. Meaning, they will not be satisfied. For any choice, any certificate, I know that a constant fraction of these terms which I'm conjuncting will evaluate to false because of the definition of PCP. That's what probability algorithm saying no means.

So it's a constant fraction of the terms are false. The terms are the things we're conjuncting over. But each term here is a constant size CNF formula. So when I and those together, I really just get one giant and of clauses. Constant fraction larger than the number of terms. And if a term is false, that means at least one of the clauses is false. And there's only a constant number of clauses in each term. So this means a constant fraction of the clauses in that giant conjunction are also false.

And that is essentially it. That is my reduction. So in the yes instance, I get 100% percent satisfiable thing. In the no instance, I get some constant strictly less than 1 satisfiable thing. Because in any solution, I get a constant fraction that turn out to be false, constant fraction of the clauses.

Now what the constant is, you'd have to work out things. You'd have to know how big your PCP algorithm is. But at least we get a constant lower bound proving-- in particular, proving there's no P [? task ?] for MAX-3SAT. This is what you might call a gap-amplifying reduction, in the sense we started with no gap. The instance of 3SAT was either true or false. And we ended up with something with a significant

gap.

So what we're going to talk about next is called gap-preserving reductions. Maybe before I get there, what we just showed is that the PCP theorem is equivalent. And in particular, we get gap problems being NP-hard. This is why we care about PCPs. And then in general, once we have these kinds of gap hardness results, we convert our-- when we're thinking about reductions from a to b, because we know gap implies inapproximability, we could say, OK, 3SAT is inapproximable, and then do, say, an l reduction from 3SAT to something else. The something else is therefore inapproximable also. That's all good.

But we can also, instead of thinking about the inapproximability and how much carries from a to b, we can think about the gap directly. And this is sort of the main approach in this lecture that I'm trying to demonstrate is by preserving the gap directly, a, well you get new gap bounds and generally stronger gap bounds. And then those imply inapproximability results. But the gap bounds are stronger than the inapproximability, and also they tend to give larger constant factors in the inapproximability results.

So what do we want out of a gap-preserving reduction? Let's say we have an instance x of A. We convert that into an instance x prime of some problem B. We're just going to think about the OPT of x versus the OPT of x prime.

And what we want for, let's say, a minimization problem is two properties. One is that the OPT-- if the OPT of x is at most some k, then the OPT of x prime is at most some k prime. And conversely. So in general, OPT may not be preserved. But let's say it changes by some prime operation. So in fact, you can think of k and k prime as functions of n. So if I know that OPT of x is at most some function of n, then I get that OPT of x prime is at most some other function of n. But there's some known relation between the two.

What I care about is this gap c. Should be a c prime here. So what this is saying is suppose I had a gap, if I know that all the solutions are either less than k or more than c times k, I want that to be preserved for some possibly other gap c prime in

the new problem. So this is pretty general, but this is the sort of thing we want to preserve.

If we had a gap of c before, we get some gap c prime after. If c prime equals c, this would be a perfectly gap-preserving reduction. Maybe we'll lose some constant factor. If c prime is greater than c, this is called gap amplification. And gap amplification is essentially how the PCP theorem is shown, by repeatedly growing the gap until it's something reasonable.

And if you want to, say, prove that set cover is log n hard, it's a similar thing where you start with a small gap constant factor, and then you grow it, and you show you can grow it to log n before you run out of space to write it in your problem essentially, before your instance gets more than polynomial. Or if you want to prove that [INAUDIBLE] can't be solved in better than whatever n to the 1 minus epsilon, then a similar trick of gap amplification works. Those amplification arguments are involved, and so I'm not going to show them here.

But I will show you an example of a gap-preserving reduction next, unless there are questions. Cool So I'm going to reduce a problem which we have mentioned before, which is MAX exactly 3 XOR- and XNOR-SAT. This is linear equations, [INAUDIBLE] two, where every equation has exactly three terms. So something like xi XOR xj XOR xk equals one, or something. You can also have negations here.

So I have a bunch of equations like that. I'm going to just tell you a gap bound on this problem, and then we're going to reduce it to another problem, namely MAX-E3-SAT. So the claim here is that this problem is one half plus epsilon, one minus epsilon, gap hard for any epsilon. Which in particular implies that it is one half minus epsilon inapproximable, unless p equals NP. But this is of course stronger. It says if you just look at instances where let's say 99% of the equations are satisfiable versus when 51% are satisfiable, it's NP-hard to distinguish between those two.

Why one half here? Because there is a one half approximation. I've kind of mentioned the general approach for approximation algorithms for SAT is take a random assignment, variable assignment. And in this case, because these

statements are about a parity, if you think of xk as random, it doesn't matter what these two are. 50% probability this will be satisfied. And so you can always satisfy at least half of the clauses because this randomized algorithm will satisfy half in expectation.

Therefore, in at least one instance, it will do so. But if you allow randomized approximation, this is a one half approximation or a two approximation, depending on your perspective. So this is really tight. That's good news. And this is essentially a form of the PCP theorem. PCP theorem says that there's some algorithm, and you can prove that in fact there is an algorithm that looks like this. It's a bunch of linear equations with three terms per equation.

So let's take that as given. Now, what I want to show is a reduction from that problem to MAX-E3-SAT. So remember MAX-E3-SAT, you're given CNF where every clause has exactly three distinct literals. You want to maximize the number of satisfied things. So this is roughly the problem we were talking about up there.

So first thing I'm going to do is I want to reduce this to that. And this is the reduction. And the first claim is just that it's an L-reduction. So that's something we're familiar with. Let's think about it that way. Then we will think about it in a gap-preserving sense.

So there are two types of equations we need to satisfy, the sort of odd case or the even case. Again, each of these could be negated. I'm just going to double negate means unnegated over here. So each equation is going to be replaced with exactly four clauses in the E3-SAT instance. And the idea is, well, if I want the parity of them to be odd, it should be the case that at least one of them is true. And if you stare at it long enough, also when you put two bars in there, I don't want exactly two of them to be true.

That's the parity constraint. If this is true, all four of these should be true. That's the first claim, just by the parity of the number of bars. There's either zero bars or two bars, or three positive or one positive. That's the two cases. And in this situation where I want the parity to be even, even number of trues, I have all the even

number of trues cases over here. Here are two of them even, and here none of them even.

And again, if this is satisfied, then all four of those are satisfied. Now, if these are not satisfied, by the same argument you can show that at least one of these is violated. But in fact, just one will be violated. So for example, so this is just a case analysis. Let's say I set all of these to be zero, and so their XOR is zero and not one. So if they're all false, then this will not be satisfied, but the other three will be.

And in general, because we have, for example, xi appearing true and false in different cases, you will satisfy three out of four on the right when you don't satisfy on the left. So the difference is three versus four. When these are satisfied, you satisfy four on the right. When they're unsatisfied, you satisfy three on the right. That's all. Claiming.

So if the equation is satisfied, then we get four in the 3SAT instance. And if it's unsatisfied, we turn out to get exactly three. So I want to prove that this is an L-reduction. To prove L-reduction, we need two things. One is that the additive gap, if I solve the 3SAT instance and convert it back into a corresponding solution to MAX-E3 XNOR SAT, which don't change anything.

The variables are just what they were before. That the additive gap from OPT on the right side is at most some constant times the additive gap on the left side, or vice versa. In this case, the gap is exactly preserved because it's four versus three over here. It's one versus zero over here. So additive gap remains one. And that is called beta, I think, in L-reduction land.

So this was property two in the L-reduction. So the additive error in this case is exactly preserved. So there's no scale. Beta equals one. If there's some other gap, if it was five versus three, then we'd have beta equal two.

Then there was the other property, which is you need to show that you don't blow up OPT too much. We want the OPT on the right hand side to be at most some constant times OPT on the left hand side. This requires a little bit more care

because we need to make sure OPT is linear, basically. We did a lot of these arguments last lecture. Because even when you don't satisfy things, you still get points.

And the difference between zero and three is big ratio. We want that to not happen too much. And it doesn't happen too much because we know the left hand side OPT is at least a half of all clauses. So it's not like there are very many unsatisfied clauses. At most, half of them are unsatisfied because at least half are satisfiable in the case of OPT.

So here's the full argument. In general, OPT for the 3SAT instance is going to be four times all the satisfiable things plus three times all the unsatisfiable things. This is the same thing as saying the-- sorry. You take three times the number of equations. Every equation gets three points for free. And then if you also satisfy them, you get one more point.

So this is an equation on those things, the two OPTs. And we get plus three times the number of equations. And because there is a one half approximation, we know that number of equations is at most two times OPT. Because OPT is at least a half the number of equations. And so this thing is overall at most six plus one seven times OPT E3 XNOR. And this is the thing called alpha in L-reduction.

I wanted to compute these explicitly because I want to see how much inapproximability I get. Because I started with a tight inapproximability bound of one half minus epsilon being impossible, whereas one half is possible. It's tight up to this very tiny arbitrary additive constant. And over here, we're going to lose something. We know from L-reductions, if you were inapproximable before, you get inapproximability in this case of MAX-E3-SAT. E3

So what is the factor? If you think of-- there's one simplification here relative to what I presented before. A couple lectures ago, we always thought about one plus epsilon approximation, and how does epsilon change. And that works really well for minimization problems. For a maximization problem, your approximation factor is-- an approximation factor of one plus epsilon means you are at least this thing times

OPT. And this thing gets awkward to work with.

Equivalently, with a different notion of epsilon, you could just think of a one minus epsilon approximation and how does epsilon change. And in general, for maximization problem, if you have one minus epsilon approximation before the L-reduction, then afterwards you will have a one minus epsilon over alpha beta. So for maximization, we had one plus epsilon. And then we got one plus epsilon over alpha beta. With the minuses, it also works out. That's a cleaner way to do maximization.

So this was a maximization problem. We had over here epsilon was-- sorry, different notions of epsilon. Here we have one half inapproximability One half is also known as one minus one half. So epsilon here is a half. And alpha was seven. Beta was one. And so we just divide by seven. So in this case, we get that MAX-E3-SAT is one minus one half divided by seven, which is 1/14. Technically there's a minus epsilon here. Sorry, bad overuse of epsilon.

This is, again, for any epsilon greater than zero because we had some epsilon greater than zero here. Slightly less than one half is impossible. So over here we get slightly less than one minus 1/14 is impossible. This is 13/14 minus epsilon, which is OK. It's a bound. But it's not a tight bound. The right answer for MAX-3SAT is 7/8. Because if you take, again, a uniform random assignment, every variable flips a coin, heads or tails, true or false.

Then 7/8 of the clauses will be satisfied in expectation. Because if you look at a clause, if it has exactly three terms and it's an or of three things, you just need at least one head to satisfy this thing. So you get a 50% chance to do it in the first time, and then a quarter chance to do it in the third time, and in general 7/8 chance to get it one of the three times. 7/8 is smaller than 13/14, so we're not quite there yet.

But this reduction will do it if we think about it from the perspective of gap-preserving reductions. So from this general L-reduction black box that we only lose an alpha beta factor, yeah we get this bound. But from a gap perspective, we can do better.

The reason we can do better is because gaps are always talking about yes instances where lots of things are satisfied. That means we're most of the time in the case where we have fours on the right hand side, or a situation where we have lots of things unsatisfied, that means we have lots of threes on the right hand side. It lets us get a slightly tighter bound. So let's do that.

So here is a gap argument about the same reduction. What we're going to claim is that 7/8 minus epsilon gap 3SAT is NP-hard, which implies 7/8 inapproximability, but by looking at it from the gap perspective, we will get this stronger bound versus the 13/14 bound.

So the proof is by a gap-preserving reduction, namely that reduction, from MAX-E3-XNOR-SAT to MAX-3SAT, E3-SAT I should say. And so the idea is the following. Either we have a yes instance or a no instance. If we have a yes instance to the equation problem, then we know that at least one minus epsilon of the equations are satisfiable. So we have one minus epsilon. Let's say m is the number of equations. In the no instance case, of course we know that not too many are satisfied. At most, one half plus epsilon fraction of the equations are satisfiable.

So in both cases, I want to see what that converts into. So in the yes instance, we get all four of those things being satisfied. So that means we're going to have at least one minus epsilon times m times four clauses satisfied. We'll also have epsilon m times three. Those are the unsatisfied. And maybe some of them are actually satisfied, but this is a lower bound on how many clauses we get.

On the other hand, in this situation where not too many are satisfied, that means we get a tighter upper bound. So we have one half plus epsilon times m times four. And then there's the rest, one half minus epsilon times three. And maybe some of these are not satisfied, but this is an upper bound on how many clauses are satisfied in the 3SAT instance versus equations in the 3x [INAUDIBLE] SAT instance.

Now I just want to compute these. So everything's times m. And over here we have four minus four epsilon. Over here we have plus three epsilon. So that is four minus epsilon m. And here we have again everything is times m. So we have 4/2, also

known as two, plus four epsilon. Plus we have 3/2 minus three epsilon. So the epsilons add up to plus epsilon. Then I check and see. Four epsilon minus three epsilon. And then we have 4/2 plus 3/2, also known as 7/2. Yes.

So we had a gap before, and we get this new gap after. When we have a yes instance, we know that there will be at least this many clauses satisfied in the 3SAT. And there'll be at most this many in the no instance. So what we proved is this bound that-- sorry, get them in the right order.

7/2 is the smaller one. 7/2 plus epsilon, comma four minus epsilon gap 3SAT, E3-SAT, is NP-hard. Because we had NP hardness of the gap before, we did this gap-preserving reduction, which ended up with this new gap, with this being for no instances, this being for yes instances. And so if we want to-- this is with the comma notation for the yes and no what fraction is satisfied. If you convert it back into the c gap notation, you just take the ratio between these two things.

And ignoring the epsilons, this is like 4 divided by 7/2. So that is 7/8 or 8/7, depending on which way you're looking. So we get also 7/8 gap. Sorry, I guess it's 8/7 the way I was phrasing it before. It's also NP-hard. And so that proves-- there's also a minus epsilon. So I should have kept those. Slightly different epsilon, but minus two epsilon, whatever. And so this gives us the 8/7 is the best approximation factor we can hope for.

AUDIENCE: In the first notation, isn't it the fraction of clauses? So between zero and one?

PROFESSOR: Oh, yeah. Four is a little funny. Right. I needed to scale-- thank you-- because the number of clauses in the resulting thing is actually 4m, not m. So everything here needs to be divided by four. It won't affect the final ratio, but this should really be over four and over four. So also known as 7/8 plus epsilon, comma one minus epsilon. Now it's a little clearer, 7/8. Cool. Yeah.

AUDIENCE: So are there any [INAUDIBLE] randomness?

AUDIENCE: So for [INAUDIBLE], you can be the randomness. Randomness would give you one half. [INAUDIBLE] algorithm gives you 1.8.

PROFESSOR: So you can beat it by a constant factor. Probably not by more than a constant factor. MAX CUT is an example where you can beat it. I think I have the Goemans Williamson bound here. MAX CUT, the best approximation is 0.878, which is better than what you get by random, which is a half I guess. Cool. All right.

Cool. So we get optimal bound for MAX-E3-SAT, assuming an optimum bound for E3-XNOR-SAT, which is from PCP. Yeah.

AUDIENCE: So I'm sorry, can you explain to me again why we don't get this from the L-reduction, but we do get it from the gap argument, even though the reduction is the same reduction?

PROFESSOR: It just lets us give a tighter argument in this case. By thinking about yes instances and no instances separately, we get one thing. Because this reduction is designed to do different things for yes and no instances. Whereas the L-reduction just says generically, if you satisfy these parameters alpha and beta, you get some inapproximability result on the output, but it's conservative. It's a conservative bound. If you just use properties one and two up here, that's the best you could show. But by essentially reanalyzing property one, but thinking separately about yes and no instances-- this held for all instances. We got a bound of seven. But in the yes and the no cases, you can essentially get a slightly tighter constant.

All right. I want to tell you about another cool problem. Another gap hardness that you can get out of PCP analysis by some gap amplification essentially, which is called label cover. So this problem takes a little bit of time to define. But the basic point is there are very strong lower bounds on the approximation factor.

So you're given a bipartite graph, no weights. The bipartition is A, B. And furthermore, A can be divided into k chunks. And so can B. And these are disjoint unions. And let's say size of A is n, size of B is n, and size of each Ai is also the same. We don't have to make these assumptions, but you can. So let's make it a little bit cleaner.

So in general, A consists of k groups, each of size n over k. B consists of k groups, each of size n over k. So that's our-- we have A here with these little groups. We have B, these little groups. And there's some edges between them. In general, your goal is to choose some subset of A, let's call it A prime, and some subset of B, call it B prime.

And one other thing I want to talk about is called a super edge. And then I'll say what we want out of these subsets that we choose. Imagine contracting each of these groups. There are n over k items here, and there are k different groups. Imagine contracting each group to a single vertex. This is A1. This is B3. I want to say that there's a super edge from the group A1 to the group B3 because there's at least one edge between them.

If I squashed A1 to a single vertex, B3 down to a single vertex, I would get an edge between them. So a super edge, Ai Bi-- Ai Bj, I should say-- exists if there's at least one edge in AI cross Bj, at least one edge connecting those groups. And I'm going to call such a super edge covered by A prime B prime if at least one of those edges is in this chosen set. So if there's at least one edge-- sorry. If this Ai cross Bj, these are all the possible edges between those groups, intersects A prime cross B prime.

And in general, I want to cover all the hyper edges if I can. So I would like to have a solution where, if there is some edge between A1 and B3, then in the set of vertices I choose, A prime and B prime in the left, they induce at least one edge from A1 to B3, and also from A2 to B3 because there is an edge that I drew here. I want ideally to choose the endpoints of that edge, or some other edge that connects those two groups. Yeah.

AUDIENCE: So you're choosing subsets A prime of A. Is there some restriction on the subset you choose? Why don't you choose all of A?

PROFESSOR: Wait.

AUDIENCE: Oh, OK. You're not done yet?

PROFESSOR: Nope. That's about half of the definition. it's a lot to say it's not that complicated of a

problem. So there's two versions. That's part of what makes it longer. We'll start with the maximization version, which is called Max-Rep. So we have two constraints on A prime and B prime. First is that we choose exactly one vertex from each group. So we got A prime intersect Ai equals one, and B prime intersect Bj equals one, for all i and j. OK And then subject to that constraint, we want to maximize the number of covered super edges.

Intuition here is that those groups are labels. And there's really one super vertex there, and you want to choose one of those labels to satisfy the instance. So here you're only allowed to choose one label per vertex. We choose one out of each of the groups. Then you'd like to cover as many edges as you can. If there is an edge in the super graph from Ai to Bj, you would like to include an induced edge. There should actually be an edge between the label you assign to Ai and the label you assign to Bj. That's this version.

The complementary problem is a minimization problem where we switch what is relaxed, what constraint is relaxed, and what constraint must hold. So here we're going to allow multiple labels for each super vertex, multiple vertices to be chosen from each group. Instead we force that everything is covered. We want to cover every super edge that exists. And our goal is to minimize the size of these sets, A prime plus B prime. So this is sort of the dual problem. Here we force one level per vertex. We want to maximize the number of covered things. Here we force everything to be covered. We want to essentially minimize the number of labels we assign.

So these problems are both very hard. This should build you some more intuition. Let me show you a puzzle which is basically exactly this game, designed by MIT professor Dana Moshkovitz.

So here's a word puzzle. Your goal is to put letters into each of these boxes-- this is B, and this is A-- such that-- for example, this is animal, which means these three things pointed by the red arrows, those letters should concatenate to form an animal, like cat. Bat is the example. So if I write B, A, and T, animal is satisfied

perfectly. Because all three letters form a word, I get three points so far.

Next let's think about transportation. For example, cab is a three-letter word that is transportation. Notice there's always three over here. This corresponds to some regularity constraint on the bipartite graph. There's always going to be three arrows going from left to right for every group. So transportation, fine. We got C-A-B. That is happy. We happen to reuse the A, so we get three more points, total of six.

Furniture, we have B, blank, and T left. This is going to be a little harder. I don't know of any furniture that starts with B and ends with T and is three letters long. But if you, for example, write an E here, that's pretty close to the word bed, which is furniture. So in general, of course, each of these words corresponds to a set of English words. That's going to be the groups on the left.

So this Ai group for furniture is the set of all words that are furniture and three letters long. And then for each such choice on the left, for each such choice on the right, you can say is, are they compatible by either putting an edge or not. And so this is-- we got two out of three of these edges. These two are satisfied. This one's not. So we get two more points for a total of eight. This is for the maximization problem. Minimization would be different.

Here's a verb, where we almost get cry, C-B-Y. So we get two more points. Here is another. We want a verb. Blank, A, Y. There are multiple such verbs. You can think of them. And on the other hand, we have a food, which is supposed to be blank, E, Y. So a pretty good choice would be P for that top letter. Then you get pay exactly and almost get pea.

So a total score of 15. And so this would be a solution to Max-Rep of cost 15. It's not the best. And if you stare at this example long enough, you can actually get a perfect solution of score 18, where there are no violations. Basically, in particular you do say here and get soy for food.

AUDIENCE: So the sets on the right are 26 letters?

PROFESSOR: Yes. The Bis here are the alphabet A through Z, and the sets on the left are a set of

words. And then you're going to connect two of them by an edge if that letter happens to match on the right, [INAUDIBLE] letter. So it's a little-- I mean, the mapping is slightly complicated. But this is a particular instance of Max-Rep.

So what-- well, we get some super extreme hardness for these problems. So let's start with epsilon, comma one gap Max-Rep is NP-hard. So what I mean by this is in the best situation, you cover all of the super edges. So the one means 100% of the super edges are covered. Epsilon means that at most an epsilon fraction of them are covered. So that problem is NP-hard.

This is a bit stronger than what we had before. Before we had a particular constant, comma one or one minus epsilon or something. Here, for any constant epsilon, this is true. And there's a similar result for Min-Rep. It's just from one to one over epsilon. So this means there is no constant factor approximation. Max-Rep is not in APX. But it's worse than that. We need to assume slightly more.

In general, what you can show, if you have some constant, p, or there is a constant p, such that if you can solve this gap problem, one over p to the k, so very tiny fraction of things satisfied versus all of the super edges covered, then NP can be solved in n to the order k time. So we haven't usually used this class. Usually we talk about p, which is the union of all these for constant k. But here k doesn't have to be a constant. It could be some function of n. And in particular, if p does not equal NP, then k constant is not possible. So this result implies this result.

But if we let k get bigger than a constant, like log-log n or something, then we get some separation between-- we get a somewhat weaker statement here. We know if p does not equal NP, we know that NP is not contained in p. But if we furthermore assume that NP doesn't have subexponential solutions, and very subexponential solutions, then we get various gap bounds inapproximability on Max-Rep.

So a reasonable limit, for example, is that-- let's say we assume NP is not in n to the polylog n. n to the polylog n is usually called quasi-polynomial. It's almost polynomial. Log n is kind of close to constant-- ish. This is the same as two to the polylog n, n to the polylog n. But it's a little clearer.

This is obviously close to polynomial, quite far from exponential, which is two to the n, not polylog. So very different from exponential. So almost everyone believes NP does not admit quasi-polynomial solutions. All problems in NP would have to admit that. 3SAT, for example, people don't think you can do better than some constant to the n.

Then what do we get when we plug in that value of k? That there is a no 1/2 to the log to the one minus epsilon n approximation. Or also, the same thing, gap is hard. Now, it's not NP-hard. But it's as hard as this problem. If you believe this is not true, then there will be no polynomial time algorithm to solve this factor gap Max-Rep.

So this is very large. We've seen this before in this table of various results. Near the bottom, there is a lower bound of two to the log to one minus epsilon n. This is not assuming p does not equal NP. It's assuming this statement, NP does not have quasi-polynomial algorithms. And you see here our friends Max-Rep and Min-Rep, two versions of label cover.

So I'm not going to prove these theorems. But again, they're PCP style arguments with some gap boosting. But I would say most or a lot of approximation lower bounds in s world today start from Max-Rep or Min-Rep and reduce to the problem using usually some kind of gap-preserving reduction. Maybe they lose the gap, but we have such a huge gap to start with that even if you lose gap, you still get pretty good results.

So a couple of quick examples here on the slides. Directed Steiner forest. Remember, you have a directed graph, and you have a bunch of terminal pairs. And you want to, in particular, connect via directed path some Ais and Bjs, let's say. And you want to do so by choosing the fewest vertices in this graph. So what I'm going to do, if I'm given my bipartite graph here for Min-Rep, I'm just going to add-- to represent that this is a group, I'm going to add a vertex here connect by directed edges here.

And there's a group down here, so I'm going to have downward edges down there.

And whenever there's a super edge from, say, A2, capital A2 to capital B1, then I'm going to say in my directed Steiner forest problem, I want a path from little a2 to little b1. So in general, whenever there's a super edge, I add that constraint.

And then any solution to directed Steiner forest will exactly be a solution to Min-Rep. You're just forcing the addition of the Ais and Bis. It's again an L-reduction. You're just offsetting by a fixed additive amount. So your gap OPT will be the same. And so you get that this problem is just as hard as Min-Rep.

Well, this is another one from set cover. You can also show node weighted Steiner trees. Log n hard to approximate. That's not from Min-Rep, but threw it in there while we're on the topic of Steiner trees.

All right. I want to mention one more thing quickly in my zero minutes remaining. And that is unique games. So unique games is a special case of, say, Max-Rep, or either label cover problem, where the edges in Ai cross Bj form a matching. For every choice in the left, there's a unique choice on the right and vice versa that matches. Well, there's at most one choice, I guess.

And I think that corresponds to these games. Once you choose a word over here, there's unique letter that matches. The reverse is not true. So in this problem, it's more like a star, left to right star. Once you choose this word, it's fixed what you have to choose on the right side. But if you choose a single letter over here, it does not uniquely determine the word over here.

So unique games is quite a bit stronger. You choose either side, it forces the other one, if you want to cover that edge. OK So far so good. Unique games conjecture is that the special case is also hard. Unique games conjecture is that epsilon one minus epsilon gap unique game is NP-hard. Of course, there are weaker versions of this conjecture that don't say NP-hard, maybe assuming some weaker assumption that there's no polynomial time algorithm.

Unlike every other complexity theoretic assumption I have mentioned in this class, this one is the subject of much debate. Not everyone believes that it's true. Some

people believe that it's false. Many people believe-- basically people don't know is the short answer. There's some somewhat scary evidence that it's not true. There's slightly stronger forms of this that are definitely not true, which I won't get into. There is a subexponential algorithm for this problem. But it's still up in the air.

A lot of people like to assume that this is true because it makes life a lot more beautiful, especially from an inapproximability standpoint. So for example, MAX-2SAT, the best approximation algorithm is 0.940. If you assume that unique games, you can prove a matching lower bound. That was MAX-2SAT for MAX-CUT, as was mentioned, 0.878 is the best upper bound by Goemans Williamson. If you assume unique games, then that's also tight. There's a matching this minus epsilon or plus epsilon inapproximability result. And vertex cover, two. You probably know how to do two. If you assume unique games, two is the right answer. If you don't assume anything, the best we know how to prove using all of this stuff is 0.857 versus 0.5. So it's nice to assume unique games is true.

Very cool results is if you look at over all the different CSP problems that we've seen, all the MAX-CSP problems, and you try to solve it using a particular kind of semi-definite programming, there's an STP relaxation. If you don't know STPs, ignore this sentence. There's an STP relaxation of all CSP problems. You do the obvious thing. And that STP will have an integrality gap.

And if you believe unique games conjecture, then that integrality gap equals the approximability factor, one for one. And so in this sense, if you're trying to solve any CSP problem, semi-definite programming is the ultimate tool for all approximation algorithms. Because if there's a gap in the STP, you can prove an inapproximability result of that minus epsilon.

So this is amazingly powerful. The only catch is, we don't know whether unique games conjecture is true. And for that reason, I'm not going to spend more time on it. But this gives you a flavor of this side of the field, the gap preservation approximation. Any final questions? Yeah.

AUDIENCE: If there's a [INAUDIBLE] algorithm [INAUDIBLE]?

PROFESSOR: It's fine for a problem to be slightly subexponential. It's like two to the n to the epsilon or something. So when you do an NP reduction, you can blow things up by a polynomial factor. And so that n to the epsilon becomes n again. So if you start from 3SAT where we don't believe there's a subexponential thing, when you reduce to this, you might end up putting it-- you lose that polynomial factor. And so it's not a contradiction. A bit subtle. Cool. See you Thursday.